DBDesigner: A Customizable Physical Design Tool for Vertica Analytic Database

Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary
Jaimin Dave, Sreenath Bodagala

Vertica Systems, an HP Company
150 Cambridge Park Dr, Cambridge, MA 02140, USA
{rvaradarajan,vbharathan,acary,jdave,sbodagala}@vertica.com

***Abstract*—In this paper, we present Vertica's customizable physical design tool, called the *DBDesigner (DBD)*, that produces designs optimized for various scenarios and applications. For a given workload and space budget, DBD automatically recommends a physical design that optimizes query performance, storage footprint, fault tolerance and recovery to meet different customer requirements. Vertica is a distributed, massively parallel columnar database that physically organizes data into *projections*. Projections are attribute subsets from one or more tables with tuples sorted by one or more attributes, that are replicated or segmented (distributed) on cluster nodes. The key challenges involved in projection design are picking appropriate *column sets*, *sort orders*, *cluster data distributions* and *column encodings*. To achieve the desired trade-off between query performance and storage footprint, DBD operates under three different design policies: (a) *load-optimized*, (b) *query-optimized* or (c) *balanced*. These policies indirectly control the number of projections proposed and queries optimized to achieve the desired balance. To cater to query workloads that evolve over time, DBD also operates in a *comprehensive* and *incremental* design mode. In addition, DBD lets users *override* specific features of projection design based on their intimate knowledge about the data and query workloads. We present the complete physical design algorithm, describing in detail how projection candidates are efficiently explored and evaluated using optimizer's cost and benefit model. Our experimental results show that DBD produces good physical designs that satisfy a variety of customer use cases.**

I. Introduction and Motivation

Automatic physical design is a challenging task and any rational choice must be cost based. Adding to this complexity, different customers have different requirements and expectations, bounded by their resource constraints. To deal with these challenges, we adopt a customizable approach in database design by allowing users to tailor their designs for specific scenarios and applications. View materialization and indexing [1], [2], [3] are effective techniques adopted in data warehouse and OLAP systems to improve query performance. While materialized views and indexes are good for query optimization, they usually come with an associated storage and maintenance overhead and are typically driven by a space budget. In addition, creating multiple materialized views and indexes can often incur significant load overhead and most customers are unwilling to slow down bulk loads. Hence, to meet different customer requirements, any physical database design tool should allow its users to trade off query performance and storage footprint for different applications.

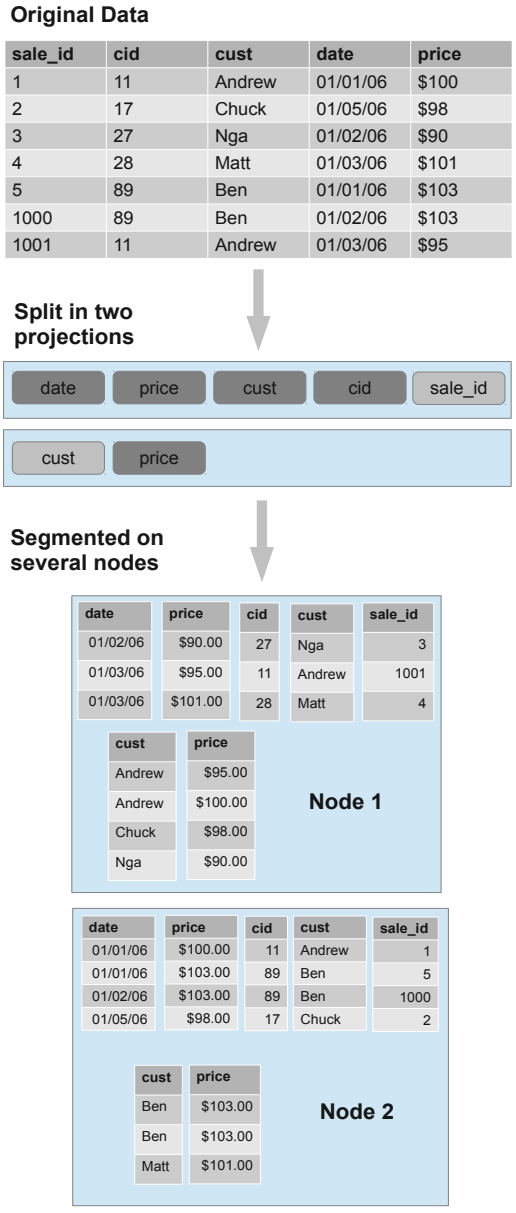


Fig. 1. Relationship between tables and projections. The $sales$ tables has 2 projections: (1) A super projection (containing all table columns), sorted by date, segmented by $HASH(sale\_id)$ and (2) a non-super projection containing (only a subset of table columns) $(cust, price)$ attributes, sorted by $cust$, segmented by $HASH(cust)$.

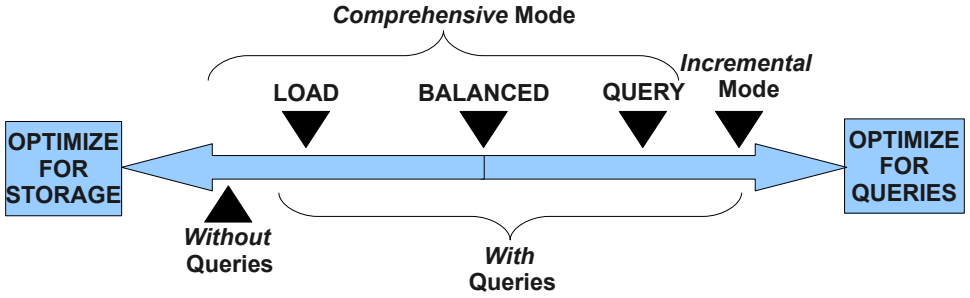


Fig. 2. Design policies and operation modes.

The Vertica Analytic Database (Vertica) [4], [5] is a distributed, massively parallel column-store RDBMS system that physically organizes table data into *projections*, which are sorted subsets of attributes that are replicated or segmented[1] on cluster nodes (as shown in Figure 1). Projections are the only physical data structure in Vertica and share similarities with materialized views [6], [7] and indexes (Section III presents more details).

In this paper, we present *DBDesigner (DBD)*, a customizable physical design tool for Vertica analytic database, that primarily operates under three design policies: (a) *load-optimized*, (b) *query-optimized* or (c) *balanced* that allow users to trade off query performance and storage footprint (as shown in Figure 2), while considering update costs. These policies indirectly control the number of projections proposed to achieve the desired balance among query performance, storage and load constraints. Under the load-optimized policy, DBD proposes the minimum required set of super projections (containing all columns) that permit fast load and delivers required fault tolerance. In the query-optimized policy, DBD may propose additional (possibly non-super) projections such that all workload queries are fully-optimized. Under the balanced policy, enough projections are proposed until a point of diminishing return is reached, in that additional projections do not bring sufficient benefits in query optimization. Section IV presents more details about the design policies.

In real-world environments, query workloads often evolve over time. A projection found helpful in the past may not be relevant today and could be wasting space slowing down loads. This space could instead be reused to create new projections that optimize current workloads. To cater to such workload changes, DBD operates in two different modes: (a) *comprehensive* and (b) *incremental* modes (as shown in Figure 2). In the comprehensive mode, DBD creates a new physical design that optimizes the current workload, while retaining parts of the existing design that are beneficial and dropping parts that are non-beneficial. In the incremental mode, customers are allowed to optionally create additional projections that optimize new queries without disturbing the existing physical design, under the assumption that workloads have not changed significantly. With no input queries, DBD optimizes purely for storage and load purposes. Section IV presents more details about the design modes and other customizable aspects of the design, including fault tolerance, recovery and feature overrides (at query and table level).

The key challenges involved in the projection design are picking appropriate column sets, sort orders, cluster data distributions and column encodings that optimize query performance while reducing space overhead and allowing faster recovery. The database designer proceeds in two major sequential phases. During the *query optimization* phase, DBD chooses projection columns, sort orders and cluster distributions (*segmentation*) that optimize query performance. During this phase, DBD enumerates candidate projections after extracting interesting column subsets by analyzing query workload for predicate, join, group-by, order-by and aggregate columns. Run length encoding (RLE) [5], [4] is given special preference for columns appearing early in the sort order, because it is beneficial for both query performance and storage optimization. Vertica's optimizer is then invoked for each workload query and given a choice of the candidate projections. A weighted benefit value associated with each candidate projection is determined based on the resulting plan, which is used to choose the best projections from among the candidates, progressively narrowing the set of candidates until a stopping condition (based on the design policy) is reached. Query and table filters are applied during this process to filter one or more queries that are sufficiently optimized by chosen projections or tables that have reached a target number of projections set by the design policy. DBD's direct use of the optimizer's cost and benefit model guarantees that it remains synchronized as the optimizer evolves over time.

During the *storage optimization* phase, DBD finds the best non-RLE column encoding schemes that achieve the least storage footprint for the designed projections via a series of empirical encoding experiments on the sample data. In addition, DBD creates the required number of *buddy* projections containing the same data but distributed differently across the cluster such that no record is stored on the same node by the buddy projections, enabling the design to be tolerant to node-down scenarios. When a node is down, buddy projections are employed to source the missing data in the down nodes. In Vertica, identical buddy projections (with same sort orders and column encodings) enable faster recovery by facilitating

[1] *Segmentation* refers to inter-node horizontal partitioning that splits tuples among computation nodes.

```
SELECT C.city, MAX(O.totalprice)
FROM customers C, orders O
WHERE
C.custkey = O.custkey AND
C.nation = 'US' AND
O.orderdate > '2012-05-01'
GROUP BY C.city;
```

Fig. 3. Example Workload Query 1.

a direct copy of their physical storage structures and DBD automatically produces such designs. Section V presents the complete physical design algorithm.

Consider a workload query (shown in Figure 3) that computes the most expensive order per US city since May 1st, 2012. Table I lists the set of interesting columns extracted from workload queries, while Tables II and III list the set of candidate cluster data distributions and sort orders explored by DBD and their associated benefits. Vertica uses segmentation to perform fully local distributed joins and efficient distributed aggregations, which is particularly effective for the computation of high-cardinality distinct aggregates. For example, a candidate projection on *customers* and *orders* table segmented by *custkey* enables a *fully distributed join* across the cluster. Similarly, a candidate projection on *customers* table segmented by *city* enables a *fully distributed group-by* operation. A candidate *customers* projection sorted by *nation* followed by *custkey* enables an efficient equality predicate evaluation on *nation* followed by a *merge join* on *custkey*, if there is an *orders* projection sorted by *custkey* as well. Optimizer's cost and benefit model is used to decide the best candidates (Section V presents more details about the physical design algorithm). Table IV lists the candidate column encodings (as defined in [4]) tried for all query columns in both tables, based on column data types and other properties.

Type	Customers Table	Orders Table
Join Cols	{custkey}	{custkey}
Group-by Cols	{city}	-
Equality predicates	{nation}	-
Inequality predicates	-	{orderdate}

TABLE I
INTERESTING COLUMN SETS FROM QUERY 1.

Table(s)	Segmentation cols	Associated Benefit
Customers, Orders	{custkey}	A *fully distributed join* on custkey
Customers	{city}	A *fully distributed group-by* on city

TABLE II
CANDIDATE DATA DISTRIBUTIONS AND THEIR BENEFITS FOR QUERY 1.

The contributions of this paper are the following:

- We present a customizable physical database design tool, *DBD*, that works with a set of configurable input parameters that allow users to trade off query performance, storage footprint, fault tolerance and recovery time to meet their requirements and optionally override design features.
- We present the complete physical design algorithm under different policies and modes. In particular, we describe in detail how physical design candidates are efficiently explored and evaluated based on the optimizer's cost and benefit model.
- We experimentally evaluate our tool to demonstrate that it can quickly provide good physical design recommendations that satisfy users' requirements.

Table	Sort Orders	Associated Benefit
Customers	<nation, custkey, ...>	*RLE-predicate* evaluation on nation, *Merge-join* on custkey
Customers	<nation, city, ...>	*RLE-predicate* evaluation on nation, *Group-by pipeline* on city
Orders	<orderdate, ...>	*RLE-predicate* evaluation on orderdate
Orders	<custkey, ...>	*Merge-join* on custkey

TABLE III
CANDIDATE SORT ORDERS AND THEIR BENEFITS FOR QUERY 1.

Table	Columns	Candidate Encodings
Customers	city	Auto, RLE
Customers	custkey	Auto, Delta-Value, Compressed-Common-Delta
Orders	totalprice	Auto, Block-Dictionary, Compressed-Delta-Range
Orders	custkey	Auto, Delta-value, Compressed-Common-delta

TABLE IV
CANDIDATE COLUMN ENCODINGS FOR QUERY 1.

The rest of the paper is organized as follows. Section II presents the related work in the area of automatic physical database design. Section III presents Vertica's data model and on-disk storage. Section IV presents the customizable aspects of the database designer. Section V presents the complete physical design algorithm in two sequential phases. Section VI presents the performance evaluation of the algorithm. Finally, Section VII presents our conclusions and future work.

II. RELATED WORK

A lot of research work has been done in automatic physical design for row-oriented databases [8], [9], [1], [3], [10], [11], [2], [12]. Most recently, Rasin et al. [13] describe the unique physical design challenges in column-store databases which include determining column sort orders, encoding and compression schemes, materialized join views and specialized approaches required in tackling column-store design problems. In [13], the authors propose an automatic physical database designer based on a cost function that involves data retrieval (I/O) and insertion costs. Their objective is to determine a set of materialized views (MVs) that minimizes the cost function and improves query performance within storage constraints. The final selection of MVs is made by formulating an Integer Linear Programming (ILP) problem, and solving it with a commercial ILP solver, similar to previous works [14] [15].

Our work is different in several aspects. First, Vertica's DBD works tightly in conjunction with Vertica's *Optimizer*

(OPT) subsystem to assess the quality of candidate projections. We use a more comprehensive cost and benefit model which considers I/O, CPU, memory and network costs of executing the training queries (discussed in Section V). It is important for any database design tool to always remain in sync with the query optimizer, because the optimizer is ultimately responsible for picking the right set of materialized views and indexes for query execution. Second, DBD enumerates multiple data distribution policies based on the query workload and evaluates the potential benefits of candidate projections in distributed scenarios, which is a complex and highly relevant problem in large-scale distributed databases. Third, DBD is capable of operating in an incremental design mode, which leverages previous design efforts and proposes improvements for more recent query workloads. Incremental design is particularly important at terabyte/petabyte data scale in lessening resource consumption in a production environment.

Abadi et al. [16] use a simple linear cost model that estimates the number of rows a query would have to process. In contrast, we use a more comprehensive cost model that considers I/O, CPU, memory and network costs of executing queries. Garcia-Alvarado et al. [17] optimize data placement in an MPP database with the objective of minimizing data movements for a given workload, under a strict constraint that only one distribution policy is allowed per table. We allow multiple data distributions per table (one per projection), giving us better opportunities for optimizing multiple query workloads. Rizzi et al. [18] try to determine which fractions of the available global space should be devoted to views and indexes, given a space constraint. In particular, they present a comparative evaluation of the benefit brought by view materialization and indexing for a single query expressed on a star schema and determine the effective trade-off between the two space fractions for the core workload. A mechanism for dynamically reorganizing data in the column store context is presented in [19], where data is partially sorted based on what is accessed by the query workload. In constrast, DBD provides customizable design modes and policies to cater to changing user requirements and query workloads.

III. Data Model and On-disk Storage in Vertica

Vertica models user data as tables of columns (attributes), although the data is not physically arranged in this manner. Data is physically stored as *projections*, which are attribute subsets from one or more tables with tuples sorted by one or more attributes. Each projection has a specific sort order on which the data is totally sorted as shown in Figure 1. Vertica requires at least one *super projection* containing every column of the anchoring table. Each projection contains *all* rows in the table, but not necessarily all columns. Any number of projections with different sort orders and subsets of the table columns are allowed. In practice, most customers have one super projection and between zero and three narrow, non-super projections. Vertica is a column-store database where each column may be independently retrieved as the storage is physically separate.

Projections can either be *replicated* or *segmented* on some or all cluster nodes. As the name implies, a replicated projection stores a copy of each tuple on every projection node. Segmented projections store each tuple on exactly one specific cluster node (providing a deterministic mapping of tuple value to node enabling many important optimizations). The most common choice is $HASH(col_1..col_n)$, where col_i is some suitably high-cardinality column with relatively even value distributions, commonly a primary key column (refer [4] for more details). Each projection column has a specific encoding scheme. A column may have a different encoding in each projection in which it appears. Figure 1 presents an example illustrating the relationship between tables and projections.

Vertica supports *prejoin* projections that permit joining the projections anchor table with any number of dimension tables via N:1 joins. Prejoin projections are not used as often in practice as we expected. This is because Vertica's execution engine handles joins with small dimension tables very well (using highly optimized hash and merge-join algorithms), so the benefits of a prejoin for query execution are not as significant as we initially predicted. In the case of joins involving a fact and a large dimension table or two large fact tables where the join cost is high, most customers are unwilling to slow down bulk loads to optimize such joins. In addition, joins during load offer fewer optimization opportunities than joins during query because the database knows nothing a priori about the data in the load stream. Experience has shown that the maintenance cost and additional implementation complexity of maintaining materialized views with aggregation and joins are not practical in real-world distributed systems.

System *K-safety* is a measure of fault tolerance in the database cluster. With K or fewer nodes down, the cluster is guaranteed to remain available. To achieve K-safety, DBD ensures that at least $K+1$ copies of each segment are present on different nodes such that a failure of any K nodes leaves at least one copy available. The failure of $K+1$ nodes does not necessarily prompt a database shutdown. Only when node failures actually cause data to become unavailable will the database shutdown, until the failures can be repaired and consistency restored via recovery. While it is not necessary that all buddy projections have the same sort order or encodings, it is necessary for them to have the same set of columns and segmentation with different node offsets, to ensure K-safety.

IV. Customizable Aspects of DBD Design

DBD works with a set of configurable input parameters to produce a customized physical design. These parameters have an impact on query performance, storage footprint, fault tolerance and recovery time described as follows:

1. Design fault tolerance (K-safety) and recovery: The value K represents the number of data replicas that exist in the cluster. These replicas allow other nodes to take over for failed nodes, allowing the database to continue running while still ensuring data integrity. DBD ensures that at least $K+1$ copies of each segment are present on different nodes

such that a failure of any K nodes leaves at least one copy available. This is done by creating K *buddy* projections for each designed projection, with the same set of columns and segmentation with different node offsets. For faster recovery, DBD automatically proposes identical buddy projections with the same sort orders and column encodings. When recovering a projection with an identical buddy from scratch, recovery may be able to directly copy the physical storage structures, enabling a speedy recovery. Alternatively, if optimizing query performance and storage footprint are the top priorities, DBD can be tuned to produce different sort orders for buddy projections.

2. Query-level design overrides: DBD provides the following query-level design overrides:
(a) Operation overrides: Produce a customized design that forces *group-by pipeline* (*sort* on *group-by* columns), fully distributed group-by (*segment* on *group-by* columns), merge join (sort on join columns) and fully distributed join (*segment* on *join* columns).
(b) Query weights: Input workload queries may be assigned weights in [0, 1] to distinguish more important queries over others. This causes DBD to prioritize candidate projections that optimize such queries when selecting the winner projections. When there are no input queries, DBD focuses on storage optimizing all input tables.

3. Table-level design overrides: DBD provides the following table-level design overrides:
(a) Data distribution: While DBD heuristically determines which tables to replicate or segment based on their relative sizes and other properties (e.g., fact vs. dimension), users can optionally force segmentation or replication on certain design tables (based on their knowledge about the data).
(b) Sort order: Produce a customized design that forces a specific sort order on a table for gaining query performance or storage benefits.

4. Design policies: Design policies have an impact on query performance and storage footprint achieved by the design. The desired balanced is indirectly achieved by controlling the number of projections proposed. DBD operates under the following policies:
(a) Load-optimized policy: Exactly $K + 1$ buddy super projections are proposed for each design table, whose sort order and segmentation would still achieve as many benefits as possible for the input query workload.
(b) Query-optimized policy: Besides $K+1$ super projections for each design table, DBD may propose additional (possibly non-super) projections until all queries referencing the design tables are *fully optimized*.
(c) Balanced policy: Enough projections are proposed until a certain percentage of queries (weighed according to their importance) have been *sufficiently optimized*. By default, this tunable parameter is heuristically set to 75%, which we found to work reasonably well on different customer datasets.

5. Design modes: To tackle variations in query workloads, DBD operates in the following modes:
(a) Comprehensive mode: In this mode, DBD is tasked with (re)designing the physical storage for the entire database from scratch for a query workload. Old or pre-existing projections that do not contribute towards the performance of query workload are dropped. This mode of operation is used when a complete initial or replacement design (justified by a significant change in the workload) is desired.
(b) Incremental mode: In this mode, DBD is tasked with determining what benefits (not provided by existing projections) might improve query performance and proposes new projections that provide these benefits to augment the existing design. This mode of operation is used when either a small set of new queries need to be optimized, or the performance of some queries optimized by a previous run of comprehensive design is inadequate.

V. The Physical Design Algorithm

When DBD is invoked with a input set of workload queries, the queries are parsed and useful query meta-data is extracted (e.g., the predicate, group-by, order-by, aggregate and join query columns). Input queries are associated with the (design) tables that they reference. A single-table query is associated with exactly one table, whereas a join query is associated with more than one table. Next, the algorithm proceeds in iterations, where in each iteration, one new projection is proposed for each table under design. The details of what an iteration does and how a projection is built and chosen by the designer, will be described in the next subsection. Once an iteration is done, queries that have been optimized by the newly proposed projections are removed, and the remaining queries serve as input to the next iteration. If a design table has reached its targeted number of projections (decided by the design policy), it is not considered in future iterations to ensure that no more projections are proposed for it. This process is repeated until there are no more design tables or design queries are available to propose projections for.

For example, suppose we have four input workload queries $Q1$, $Q2$, $Q3$ and $Q4$ (to optimize for) referencing tables A, B, and C as follows:

- $Q1$ is a selection query on table A.
- $Q2$ is a join query between table A and B.
- $Q3$ is a join query between table A and C.
- $Q4$ is a group-by query on table A.

Under the load-optimized design policy, the flow of a designer invocation may go as follows. In the first iteration, a super projection is proposed respectively for A, B and C. If the super projections for A and B optimize for $Q1$ and $Q2$, we remove $Q1$ and $Q2$ from the input query set. If, for example, the super projection for C does not optimize any queries, it is removed from the list of proposed projections (we only keep projections that are beneficial to some input query). In the second iteration, again a super projection is proposed for each design table. The super projections for A and C this time optimize for query $Q3$, which is then removed from the query

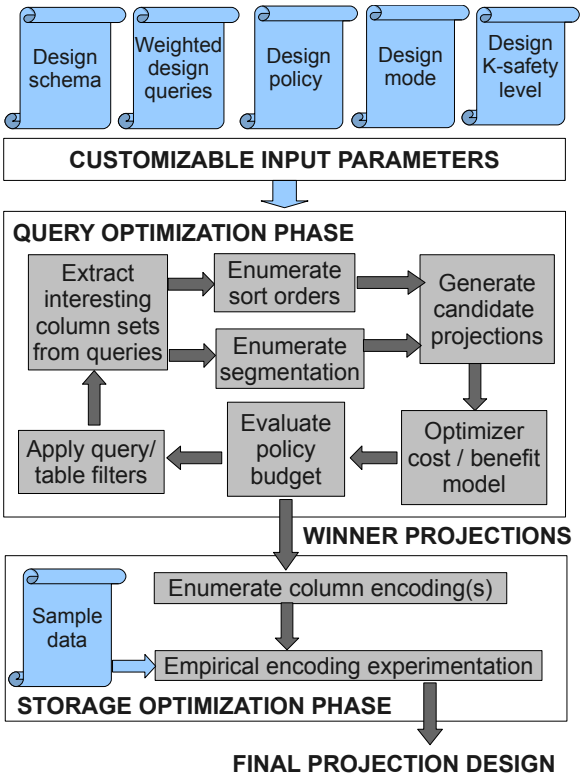


Fig. 4. DBDesigner Architecture.

set. At this point, all queries referencing B and C have been handled, so we remove them from the set of design tables (after making all the proposed projections K-safe). Under the query-optimized design policy, we propose an additional projection for A to optimize for $Q4$. In the third iteration, a non-super projection is proposed for A to optimize for $Q4$, finishing the design process (after making the proposed projection K-safe again).

In order to form the complete search space for enumerating projections, we identify the following design features in a projection definition.

- Sort order (feature 1).
- Segmentation (feature 2).
- Column encoding schemes (feature 3).
- Column sets (select columns) (feature 4).

All possible combination of choices in the above features, forms the complete search space of projections. Because an exhaustive search is infeasible, we explore a portion of the search space described as follows. We enumerate choices for features 1 and 2 above, and use optimizer's cost and benefit model to compare and evaluate them (during the *query optimization phase*). Note that the choices made for features 3 and 4 typically do not affect the query performance significantly. The winners decided by the cost and benefit model are then extended to full projections by filling out the choices for features 3 and 4, which have a large impact on load performance and storage (during the *storage optimization* phase). Figure 4 presents the architecture of DBDesigner.

A. Query Optimization Phase

We will present the candidate enumeration process, after covering its preliminaries.

Sort Order: A *sort order* (SO) consists of the following components:

- A non-empty list of sort columns, referred to as SC.
- A possibly empty list of *RLE-friendly* columns, referred to as RC. We say that a list of columns L from table T is RLE-friendly if the number of RLE buckets defined by sorting and run-length encoding L does not exceed a predefined threshold. Let function $LRC(L)$ compute and return the longest prefix of L that is RLE-friendly (depending on the grouping cardinality of columns in L, considering correlations).

For example, if $SO = < a, b, c >$, then $SO.SC = < a, b, c >$, and $SO.RC$ could be $<>$, $< a >$, $< a, b >$ or $< a, b, c >$.

SetList: A *setlist* object (SL) is a list of sets of columns. For example, let setlist object $SL = < \{a\}, \{b, c\}, \{d\}, \{e, f\} >$. When a set is a singleton set, we omit the curly braces. So SL in the previous example can also be written as $< a, \{b, c\}, d, \{e, f\} >$. By definition, a list element in a setlist object is a set of columns. For example, $\{e, f\}$ is a list element of SL. The length of a setlist object, SL denoted as $|SL|$, is the number of list elements in it. A setlist object with a single list element (i.e., $|SL|$=1) is a *singleton* setlist. For example, $< a >$ and $< \{a, b\} >$ are both singletons. A setlist object is a compact representation of a class of lists. A list is derived from a setlist, SL as follows. For each list element, S in SL, replace S with a particular permutation of columns in S. Continuing the above example, SL represents four lists: $L1 = < a, b, c, d, e, f >$, $L2 = < a, c, b, d, e, f >$, $L3 = < a, b, c, d, f, e >$, $L4 = < a, c, b, d, f, e >$. In other words, we say all these four lists can be derived by SL. The function that computes the set of lists derived from a setlist object is referred to as $ExpandSL$. In the above example, $ExpandSL(SL) = \{L1, L2, L3, L4\}$. In practice, to minimize the number of candidate sort orders, $ExpandSL$ returns only the best permutation per setlist that maximizes RLE compression.

Sort Order Class: A *sort order class* object (SOC), is a non-empty setlist object, which represents a class of sort orders. We say a sort order SO, is derived from a SOC, if $SO.SC$ belongs to $ExpandSL(SOC)$. For example, let a query Q have a group-by operation on columns a, b and c. The SOC object generated from the group-by operation of Q, is $< \{a, b, c\} >$. A total of six sort orders can be derived from SOC.

*1) **Extracting Interesting Column Sets from Workload Queries**:* To prune the search space, only interesting $SOCs$ defined based on the physical properties of queries are generated, as follows.

Predicate columns (PRED): In a conjunctive query Q, all predicates referring to a single table within a conjunct, generate a singleton SOC. For example, consider the following conjunctive predicates for tables $T1$ and $T2$: WHERE

($T1.y = 20$ OR $T1.z > 30$) AND ($T2.w < 40$ OR $T2.x = 10$). It generates an interesting $SOC < \{y, z\} >$ for table $T1$ and $< \{w, x\} >$ for table $T2$. Similarly, a query with predicates of the form WHERE ($T1.a = 20$ OR $T2.b <= 7$ OR $T1.c > 12$ OR $T2.d = 100$) generates $SOCs < \{a, c\} >$ and $< \{b, d\} >$ for tables $T1$ and $T2$. For the query shown in Figure 3, $< C.nation >$ and $< O.orderdate >$ are generated for *customers* and *orders* tables respectively.

Group-by columns (GBY): Because there is *no* specific preference of order required by a group-by operation within a *set* of group-by columns, for a query Q with G being its set of group-by columns, a singleton SOC of the form $< \{G\} >$ is generated. For example, a group-by operation, GROUP BY $T1.y$, $T1.z$ generates a $SOC < \{y, z\} >$. For the query in Figure 3, a $SOC < C.city >$ is generated for the *customers* table.

Order-by columns (OBY): Since there *is* a specific preference of order required by an order-by operation, for a query Q with O being its *list* of order-by columns, a SOC of the form $< O >$ is generated. For example, an order-by operation, ORDER BY $T1.x$, $T1.y$ generates a $SOC < x, y >$.

Join columns (JOIN): Join columns are similar to group-by columns in that no specific order preference is required among join columns from an input table. Let table T be one of the inputs to a join operator and S be the set of join columns from T. This generates a singleton SOC of the form $< \{S\} >$. For example, a join operation between $T1$ and $T2$ such that ($T1.y = T2.a$ AND $T1.z = T2.b$), generates a $SOC < \{y, z\} >$ for table $T1$ and a $SOC < \{a, b\} >$ for table $T2$. For the query in Figure 3, a $SOC < O.custkey >$ for orders table and a $SOC < C.custkey >$ for *customers* table are generated.

Aggregate columns (AGG): Each aggregate column generates a singleton SOC. For the query in Figure 3, a $SOC < O.totalprice >$ for orders table is generated.

Partition-by and order-by columns in analytic over clauses (OVER): For an SQL analytic function with *set* P being the partition-by columns and *list* L being the order-by columns in its over clause, a SOC of the form $< \{P\}, L >$ is generated. For example, an analytic function RANK() over (PARTITION BY $city$, $month$ ORDER BY $sales$) generates a $SOC < \{city, month\}, sales >$.

2) ***Enumerate Candidate Sort Orders****:* We enumerate $SOCs$ bottom up, starting with an initial set of $SOCs$ referred to as *seeds*. During the enumeration process, each SOC is *refined* or *extended*. When a SOC is *refined*, the set of $ExpandSL(SOC)$ shrinks. For example, a $SOC < \{a, b, c\} >$ may be refined to a SOC' = $< b, \{a, c\} >$, where $ExpandSL(SOC')$ is a subset of $ExpandSL(SOC)$. When a SOC is *extended*, its length increases. For example, a $SOC < b, \{a, c\} >$ may be extended to SOC' = $< b, \{a, c\}, d >$. The initial $SOCs$ are referred to as *seeds* and the columns we use to refine or extend existing $SOCs$ are referred to as *extenders*.

For example, for the query shown in Figure 3, let a seed SOC S = $< C.nation >$ and an extender E = $< C.city >$. When we extend S with E to obtain a new SOC S' = $< C.nation, C.city >$. As another example, let a seed SOC S = $< \{a, b\} >$ and extender E = $< b >$. When we refine S with E, we obtain a new SOC S' = $< b, a >$. Note that *refinement* is possible only when all columns in E are *contained* in SOC (i.e., E should be a subset of SOC). Intuitively, a SOC object that represents multiple sort orders has no preferences among these sort orders. However, when it is refined by an extender that has its own preferences, the resulting SOC reflects the preference from that extender. In the previous example, SOC has no preference over sort orders $< a, b >$ or $< b, a >$, but extender E has a preference with column b being ahead of the list of sort columns.

	Extenders					
$SOCs$	EQ PRED	INEQ PRED	GBY	OBY	JOIN	OVER
EQ PRED	E, R	E, R	E, R	E, R	E, R	E, R
INEQ PRED	R	R	R	R	R	R
GBY	R	R	R	R	R	R
OBY	R	R	R	R	R	R
JOIN	R	R	R	R	R	R
OVER	R	R	R	R	R	R

TABLE V
SORT EXTENSIONS(E) AND REFINEMENTS(R) TRIED BY $SortExt$

Now we formally describe how an extender, E, is applied to a SOC for refinement or extension, to obtain a new SOC' object. We apply a *sort-extension* ($SortExt$) and an *RLE-extension* ($RLEExt$) operation in sequence. $SortExt$ attempts to merge SOC and E while retaining the sort optimizations required by both. It has stricter requirements than $RLEExt$, described as follows. $SortExt$ *extends* SOC with E only when both come from the same query and SOC originates from equality predicates. In all other cases, $SortExt$ tries only *refinements* (as shown in Table V), provided all the columns in E are contained in SOC. In a high-level, $SortExt$ tries to append E to the end of SOC. If the columns in E and SOC are disjoint, this is trivial to do. When there are overlappings, we try merging the common columns as possible. A point p is identified in SOC such that columns after p are compatible with a prefix list of columns in E. Let this common list of columns be denoted as CC. The resulting SOC' has three sections: {columns in SOC before CC}, {CC}, {columns in E after CC}. To identify point p, we try out all possible points in SOC starting from the beginning. A few examples of the $SortExt$ operation are as follows:

- $SortExt(< \{a, b, c\}, \{d, e\} >, < \{c, d, e\} >)$ = $< \{a, b\}, c, \{d, e\} >$ (refinement, CC = $< c, \{d, e\} >$).
- $SortExt(< \{x, y, z\}, \{u, v\} >, < z, x, u, v, w >)$ = $< y, z, x, u, v, w >$ (extension, CC = $< z, x, u, v >$).
- $SortExt(< \{a, b, c\}, \{d, e\} >, < c, e, d >)$ = $< \{a, b\}, c, e, d >$ (refinement, CC = $< c, e, d >$).

	Extenders				
$SOCs$	PRED	GBY	OBY	JOIN	OVER
PRED	E, R	E, R	-	E, R	-
GBY	E, R	E, R	-	E, R	-
OBY	E, R	E, R	-	E, R	-
JOIN	E, R	E, R	-	E, R	-
OVER	E, R	E, R	-	E, R	-

TABLE VI
RLE EXTENSIONS(E) AND REFINEMENTS(R) TRIED BY $RLEExt$

If $SortExt(SOC, E)$ fails to produce a new SOC', then $RLEExt$ is applied. $RLEExt$ requires that E is RLE-friendly and a singleton SOC (as shown in Table VI). A new SOC' is created as follows. For each list element, L, in SOC, split L into two list elements $\{L \cap E\}$ followed by $\{L - E\}$. At the end, a new list element containing all those columns in E that are not in SOC (if any), is appended to SOC'. Let C be a set of columns defined by taking columns from SOC', starting from the first list element up to subsequent list elements, until all columns in E have been included. We say $RLEExt$ succeeds only if C is RLE-friendly. A few examples of the $RLEExt$ operation are as follows:

- $RLEExt(< \{a, b\} >, < b >) = < b, a >$ (refinement, $C = \{b\}$).
- $RLEExt(< \{a, b\} >, < c >) = < \{a, b\}, c >$ (extension, $C = \{a, b, c\}$).
- $RLEExt(< a, \{b, c, d\}, e >, < \{b, d\} >) = < a, \{b, d\}, c, e >$ (refinement, $C = \{a, b, d\}$).

Having defined the $SortExt$ and $RLEExt$ operations, we now resume the description of the SOC enumeration process. For existing $SOCs$, we continue to apply extenders to them using $SortExt$ and $RLEExt$ operations (in sequence), until no new $SOCs$ can be generated this way. $RLEExt$ is applied only when $SortExt$ fails to generate a new SOC. Those $SOCs$ that are extended or refined are then discarded, because the newly generated ones are superior to them. We refer to a SOC that cannot be further extended or refined as a *terminal*. At the end of this repeated extension and refinement process, all remaining $SOCs$ are terminals. Next, we apply $ExpandSL$ to each terminal SOC to derive the final set of sort orders from it. To minimize the number of candidate sort orders, $ExpandSL$ returns only the best sort order per terminal that maximizes RLE compression. Algorithm 1 presents the pseudocode of sort order enumeration algorithm.

Algorithm 1 The *Sort Order Enumeration* Algorithm, *enumerateSOCs*

Input: Seed $SOCs$ for a set of design tables, T.
Output: Sort orders, SO.

```
1:  SO = φ;
2:  for each query table, T under design do
3:    terminalSOCs = φ;
4:    currentSOCs = SOCs[T];
5:    extenders = SOCs[T];
6:    while currentSOCs is non-empty do
7:      newSOCs = φ;
8:      for each SOC in currentSOCs do
9:        for each E in extenders do
10:         newSOC = SortExt(SOC, E);
11:         if newSOC is empty then
12:           newSOC = RLEExt(SOC, E);
13:         end if
14:         if newSOC is non-empty then
15:           Append newSOC to newSOCs;
16:         else
17:           Append SOC to terminalSOCs;
18:         end if
19:       end for
20:     end for
21:     currentSOCs = newSOCs;
22:   end while
23:   Append ExpandSL(terminalSOCs) to SO;
24: end for
25: return SO;
```

*3) **Enumerate Candidate Cluster Distributions**:* The segmentation candidates are generated from the *group-by* and *join* columns (extracted from the workload queries), because segmenting on such columns facilitate a *fully distributed* group-by and join operation respectively. We say a set of columns, S from table, T is *segmentation-friendly*, if the estimated number of distinct values produced by S exceeds a predefined threshold. DBD evaluates each candidate for segmentation-friendliness. If none are found to be suitable for segmentation, it automatically selects a set of high-cardinality columns with relatively even value distributions to avoid data skew across the cluster. In addition to the segmentation candidates, DBD also considers a replicated candidate cluster distribution.

*4) **Generate Candidate Projections**:* Candidate projections are generated by combining each candidate sort order with every segmentation candidate. Because it is expensive to create a large number of projections, we limit the number of candidate projections explored by DBD, per table. If a choice needs to be made among several candidate projections, we heuristically choose projections with shorter segmentation keys and longer sort orders (because they could potentially provide more plan features).

*5) **Using Optimizer's Cost Estimates for Candidate Selection**:* The candidate selection process proceeds as follows. A *virtual projection catalog* (VPC) is created out of the entire set of projection candidates derived from the enumeration process. The optimizer is invoked once per query q on VPC to obtain the query plan (we briefly explain optimizer's cost and projection benefit models later in this section). We call this the *ideal plan* for query q because it is generated in the presence of all possible projection candidates. For each candidate projection chosen by the optimizer in the generated

query plan for query q, we assign benefit points to it, where the points are proportional to the weight of query and the weights of the benefits brought by the candidate projection. We then add up the points accumulated for each candidate and select the best candidate projection per table with the highest points.

Let $P = \{p_1, p_2, ..., p_t\}$ be the set of candidate projections in VPC. Let $Q = \{q_1, q_2, ..., q_m\}$ be the input set of workload queries and $W(Q) = \{w(q_1), w(q_2), ..., w(q_m)\}$ be the set of individual query weights. Let $B = \{b_1, b_2, ..., b_n\}$ be the set of possible projection benefits and $W(B) = \{w(b_1), w(b_2), ..., w(b_n)\}$ be the set of individual benefit weights. Let $b_1, b_2, ..., b_r$ be the set of benefits brought by projection p_i to query q_j, in a chosen plan for that query. Equation 1 defines a benefit function, $b(p_i, q_j)$, that returns the total weight of the set of benefits brought by projection p_i to query q_j. Equation 2 defines the accumulated benefit of a candidate projection, p_i. Let $p_1, p_2, ..., p_c$ be the set of candidate projections in the chosen plan for query q_j. Equation 3 defines the accumulated benefit of a query q_j.

$$b(p_i, q_j) = \sum_{k=1}^{r} w(b_k) \tag{1}$$

$$Benefit(p_i) = \sum_{j=1}^{m} w(q_j) * b(p_i, q_j) \tag{2}$$

$$Benefit(q_j) = \sum_{i=1}^{c} b(p_i, q_j) \tag{3}$$

Once the best candidates per table are selected, the loser projections are dropped from VPC and the optimizer is invoked once again per query to calculate the actual benefits achieved. We call this the *actual plan* for query, q, because VPC now retains only the winner projections. The optimization ratio of a query is defined as the ratio of its benefits achieved in the actual plan to that achieved in the ideal plan (as defined in Equation 4). A query is fully optimized if its optimization ratio is very close to 1. The entire candidate enumeration and selection process is repeated for several iterations on those queries that are not fully optimized until the design policy budget is hit. Algorithm 2 presents the design algorithm. Next we briefly describe the optimizer's cost and projection benefit model.

$$OptRatio(q_j) = \frac{Benefit(q_j)\ in\ the\ actual\ plan}{Benefit(q_j)\ in\ the\ ideal\ plan} \tag{4}$$

Optimizer's cost model: One of the main challenges faced by the optimizer is to pick the correct set of projections that covers all columns needed to answer the query as efficiently as possible. We call this the *covering projection set* (CPS) of the query, defined as the set of projections (one per table), that can be used to fetch the data required from those tables. This problem is hard because if a query contains T tables with a maximum of P eligible projections per table, there are potentially P^T ways of picking a covering projection set. Since exploring the entire search space is practically impossible, optimizer prunes some projection sets that could potentially yield suboptimal plans, based on their physical properties. A *physical property* of the projection (or input stream) is a property that has the potential to make upstream operations in a query plan cheaper. Each query generates a list of physical properties that represents the set of desired features for that query. This set of query properties is used to prune the inital search space of choosing projection sets. For example, if a input stream (or projection) is sorted on a specific set of columns and a prefix of the same set is used in a group-by clause, then this physical property has the potential to make the group-by operation cheaper. A description of the join order enumeration and selection process is beyond the scope of this paper. Following is a list of physical properties tracked by the

Algorithm 2 The *Physical Design* Algorithm, *DBDesign*

Input: Queries(Q), Tables(T), Policy(P), K-Safety(K), Sample Data(S).
Output: Winner projections, $winProjs$.

```
1: winnerProjs = φ;
2: for each query q_j in Q do
3:    ICS[q_j] = extractICS(q_j); {extract query columns}
4:    optRatio[q_j] = 0;
5: end for
6: while evalPolicyBudget(winProjs, optRatio, P) do
7:    curICS = getFurtherOptimizableICS(ICS, optRatio);
8:    SOCs = enumerateSOCs(curICS);
9:    SEGs = enumerateSEGs(curICS);
10:   candProjs = enumerateProjections(SOCs, SEGs);
11:   for each query q_j in Q do
12:      idealPlan[q_j] = getOPTPlan(q_j, candProjs);
13:      for each projection p_i in idealPlan[q_j] do
14:         Compute b(p_i, q_j); {Eq 1}
15:         Use b(p_i, q_j) to update benefit[p_i] {Eq 2}
16:         Add b(p_i, q_j) to idealBenefit[q_j]; {Eq 3}
17:      end for
18:   end for
19:   for each table t in T do
20:      winner[t] = chooseHighestBenefitProj(candProjs[t]);
21:      Append winner[t] to winProjs;
22:   end for
23:   for each query q_j in Q do
24:      actualPlan[q_j] = getOPTPlan(q_j, winProjs);
25:      for each projection p_i in actualPlan[q_j] do
26:         Compute b(p_i, q_j); {Eq 1}
27:         Add b(p_i, q_j) to actualBenefit[q_j]; {Eq 3}
28:      end for
29:      optRatio[q_j] = (actualBenefit[q_j]/idealBenefit[q_j]);
30:   end for
31: end while
32: makeProjsKSafe(winProjs, K);
33: enumerateAndFindBestEncodings(winProjs, S);
34: return winProjs;
```

optimizer:
1. Sorted property: Represents the sort order needed for a merge-join, group-by pipeline or order-by operation.
2. Segmented property: Represents the data segmentation necessary to perform an operation in parallel (such as a local join or a fully distributed group-by, etc). A projection path satisfies this property, if the path is segmented on a subset of the required columns referenced in the property.
3. Replicated property: Represents the data replication that can be used to perform distributed operations in parallel (such as a local join). A projection path satisfies this property, if the path is replicated on all nodes and contains a subset of the required tables referenced in the property.
4. Sorted-Segmented property: Represents the sort order and segmentation needed for a local merge-join or a distributed group-by pipeline operation.

Optimizer keeps the lowest cost projection for each physical property. The cost of an access path is calculated based on the data that flows through the path. The final cost is a weighted sum of the estimated CPU, memory, network, disk costs and the level of parallelism involved. All access paths are evaluated based on following aspects:
1. I/O cost: The estimated amount of data to be read from disk, based on column encoding and compression scheme. For example, in an RLE-encoded column at the beginning of the sort order, the amount of data read could be much lower, based on the number of distinct values in the column.
2. Network cost: The estimated size of the data that needs to be transmitted over the network. For example, if a join requires resegmentation or broadcast, then the amount of data sent over the network can be significant.
3. Memory cost: The estimated size of data that needs to stored temporarily in memory. For example, in a group-by hash operation, it is the amount of space needed to temporarily store the hash table.
4. CPU cost: The estimated size of the data that needs to be processed. For example, in a hash join operation, it is the estimated size of inner and outer tables that needs to hashed.
5. Parallelism: The number of nodes simultaneously involved in a particular access path. For example, in a fully distributed join operation, all cluster nodes are simulateneously involved in performing the join operation.

Projection benefits model: DBD effectively uses its own credit system for mapping query plan highlights to benefit points. The highlights in a query plan are the set of benefits received by the query due to the presence of a set of projections. DBD parses the query plans produced by the optimizer and assigns points to the chosen projections according to the following highlights in the plan:
1. Group-by pipeline: Input stream sorted on group-by columns, enabling on-the-fly computation (without the requirement of an in-memory hash table).
2. Fully distributed group-by: Input stream segmented on a subset of group-by columns, enabling a local group-by operation on each cluster node.
3. Fully distributed join: Inner and outer input streams segmented on a subset of the join columns, enabling a local join operation on each cluster node.
4. Fully distributed merge join: Inner and outer input streams sorted on join columns and segmented on a subset of join columns, enabling a local merge join on each cluster node.
5. Fully distributed analytics: For an analytic operation with partition-by columns, p and order-by columns, l in its over clause, an input stream segmented by p and sorted by p, l enables a local analytic operation on each cluster node.
6. Eliminated sort operation: Input stream sorted on columns required by an upstream operator (eliminating the need for sorting).
7. Sort merge join: One of the input streams sorted on join keys, enabling a sort merge join (by sorting the other input stream).
8. Late materialization: Late materialized plan has a lower I/O cost than an early materialized plan (more details in [20]).
9. RLE predicate evaluation: RLE-encoded predicate column enabling predicate evaluation in compressed form. Benefits are calculated according to the predicate type (equality vs. inequality), sort position and the estimated cardinality of the predicate column based on correlations among sort columns.
10. RLE join and group-by operation: RLE-encoded join or group-by columns enabling the operation in compressed form.

B. Storage Optimization Phase

One of Vertica's key techniques for performance is data-specific encoding and compression. Sophisticated encoding and compression algorithms minimize storage and I/O bandwidth requirements when handling petabyte scale data sets. Rather than manually specifying encodings, which requires very scarce human experts, DBDesigner's storage optimization component automatically chooses the best techniques. While very effective, this process is usually very resource and time intensive, often requiring all of a cluster's resources for many hours to produce an optimal storage design. The key challenge in storage optimization phase involves picking the best column encoding and compression schemes (among several candidate schemes) that minimize storage footprint, as quickly as possible using minimal computing resources. Specifically, the challenges are:

- Identifying candidate encoding schemes for each column based on type, cardinality and sortedness.
- Finding the best encoding scheme among multiple candidate choices that yields the least storage footprint (via empirical storage experiments). This involves finding an appropriate data sample for encoding analysis, which includes identifying an appropriate sampling technique and sample size suitable for empirical experimentation.

Overview: Lamb et al. [4] present the list of encoding and compression schemes implemented in Vertica, including which schemes are ideal for which columns based on their type, cardinality and sortedness properties. For more details, refer Section 3.4 in [4]. DBD identifies the best column encoding schemes that minimize storage footprint by trying all ideal

encoding choices for each column on a chosen data sample and comparing their storage footprints. A column with an unambiguous encoding choice is not included in the empirical experiments unless they happen to have an impact on the storage footprint achieved by other columns (especially when they are part of projection's sort order or segmentation). DBD skips trying RLE encoding type on a high-cardinality column (identified through statistics or primary key constraints) because it is not an ideal choice for such columns.

Data sampling for encoding analysis: We experimented two different methods: (a) random sampling and (b) clustered sampling. Random sampling involves selecting a random set of rows for encoding tests. The main disadvantage is I/O performance, because it typically involves scanning many pages and applying a random function. We also noticed that using such a random sample is usually detrimental for certain encoding types like RLE because it tends to increase the apparent number of distinct values and propensity toward sequences (hurting compressed-common-delta and a few other encoding types). To overcome these disadvantages, we implemented a clustered sampling technique that selects contiguous rows from equally spaced bands in the database, based on a preselected sample size and band count. The advantages of this technique is both performance (because we only read the required number of pages) and quality of the sample is more appropriate for encoding experiments.

Encoding sample size: To decide on an appropriate sample size, we experimented with real customer datasets, trying different sample sizes in million increments and measured the footprint ratio achieved by individual design tables (presented in Equation 5 for a table, T).

$$FootprintRatio(T) = \frac{Footprint\ with\ optimal\ encodings}{Footprint\ with\ approx.\ encodings} \quad (5)$$

Optimal column encodings denote the ones found using all available data, while the approximate encodings are those selected using the data sample. Note that footprint ratio cannot be more than 1.0 and closer it is to 1.0, the better. As can be observed in all customer datasets, we found that one million clustered samples are sufficient to achieve a footprint ratio as high as 0.996. We also noticed that this sample size is sufficient for finding the best encodings in most columns, while some columns might end up having the second or third best encoding.

VI. Performance Evaluation

In this section, we present some experimental results that demonstrate the query execution times, storage footprints and design times achieved by DBD under load-optimized, query-optimized and balanced design policies. For the experiments, we used a four node cluster with two Intel Xeon X5670 processors [21], 94GB of RAM and a 4.2 TiB RAID-5 disk drive on each node.

Dataset: The customer dataset is a 1TB star schema, with the table *order_upc* being the largest fact table containing all customer orders with foreign keys to the dimension tables *product* (17-column table with product information), *location* (30-column table with purchase location information), *date* (60-column table with order date information in day, week, month, quarter and year roll-ups) and *user_product_grp* (with user group information). Due to space constraints, we omit mentioning other dimension and fact tables.

Queries: Typical workload queries were analytical in nature involving complex joins, multiple (distinct) aggregates and range predicates. Some query examples are listed as follows:

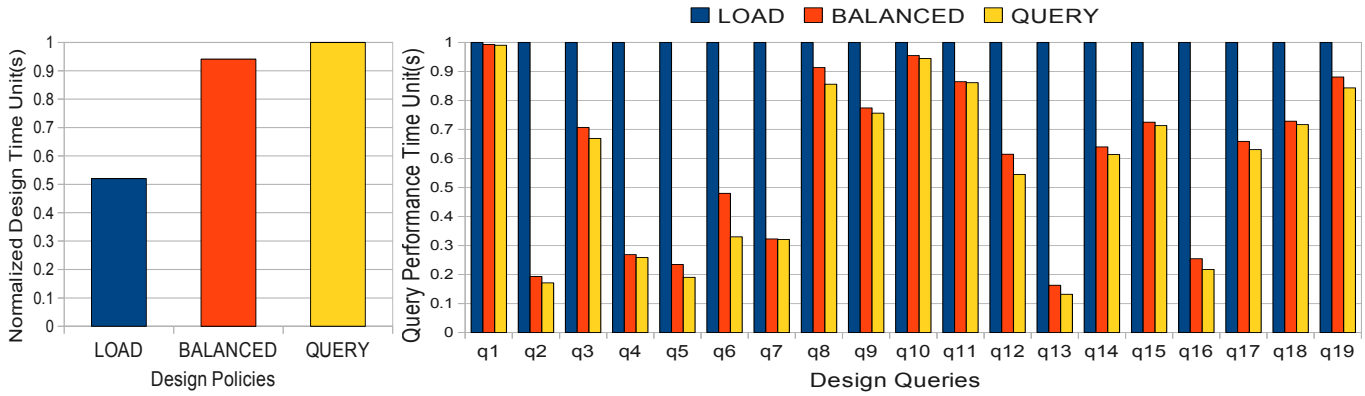
- How many products were sold in each category across a particular chain in the first quarter of 2011?
- What brand manufacturers are selling at each retailer during the most recent 6 weeks?
- How many product units and money was spent in all categories during the past 2 years, by year and division?
- What is the average number of trips made by loyalty card customers during 2010?
- How many identifiable customers were seen in the past two weeks?
- How many products were sold by category and operator-id in a year 2010?

Results: Figure 5(a) shows the design performances achieved by DBD under all policies, with design times normalized across policies. As expected, DBD under the load-optimized policy was the fastest because it is only required to come up with the minimum set of projections, while query-optimized policy was the slowest because it tries to fully optimize all workload queries. DBD under the balanced policy was tuned to sufficiently optimize at least 75% of workload queries, achieving a design performance very close to that of the query-optimized policy.

Query performances were measured after deploying the resulting design. Figure 5(b) shows the query performances achieved by DBD under all policies, with individual query times normalized across policies. A total of 19 workload queries were tested with cold caches. Query performances achieved by the query-optimized design was the best, while load-optimized design performed the worst as expected. The balanced design achieved query performances comparable to that of query-optimized design (only slightly worse), because many of the workload queries were sufficiently optimized. Figure 6 shows the normalized storage footprints achieved under different policies. As expected, the load-optimized policy achieved the least storage footprint, while the query-optimized had the largest footprint. The only exception was the *user_product_grp* table where the difference is minimal, because it is the smallest table with negligible storage footprint. Consequently, the load performance achieved by the load-optimized design was the best, while query-optimized design performed the worst (we omit presenting these results due to space constraints).

VII. Conclusions and Future Work

This paper presents Vertica's physical database design tool that is customizable for different applications. We present the complete physical design algorithm, describing in detail



(a) Design Performance

(b) Query Performance

Fig. 5. Design and query performance achieved under different policies.

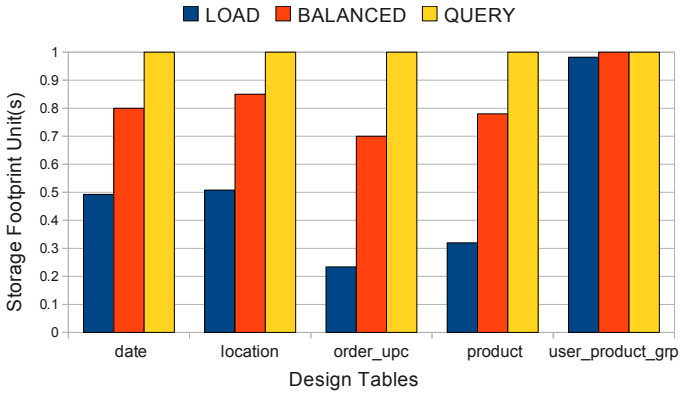


Fig. 6. Storage footprint achieved under different design policies.

how candidates are efficiently explored and evaluated by the optimizer's cost and benefit model. We experimentally evaluate our tool under different policies and demonstrate that it can quickly provide good physical design recommendations satisfying users' requirements and resource constraints. In the future, we plan to implement a daemon design mode where DBD will be coupled with our workload analyzer tool and invoked automatically in the background as workloads change significantly.

VIII. Acknowledgments

Mingsheng Hong, Priya Arun, Chang-Jian Sun, Alex Rasin and Shilpa Lawande, while not authors of this paper, designed and wrote significant portions of the Database Designer. Goetz Grafe contributed useful and insightful comments to this paper and its argument. Shalu Tiwari and Meghan Elledge were instrumental in performing the experiments. Sarah Lemaire helped in editing and proofreading this paper.

References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases," in *VLDB*. Morgan Kaufmann Publishers Inc., 2000, pp. 496–505.

[2] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "Db2 design advisor: integrated automatic physical database design," in *VLDB*. VLDB Endowment, 2004, pp. 1087–1097.

[3] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*. ACM, 2004, pp. 359–370.

[4] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The Vertica analytic database: C-store 7 years later," vol. 5, no. 12. VLDB Endowment, Aug. 2012, pp. 1790–1801.

[5] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden and E. J. O'Neil et.al, "C-Store: A Column-oriented DBMS," in *VLDB*, 2005, pp. 553–564.

[6] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance," in *VLDB*, 1991, pp. 577–589.

[7] M. Staudt and M. Jarke, "Incremental Maintenance of Externally Materialized Views," in *VLDB*, 1996, pp. 75–86.

[8] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for microsoft sql server," in *VLDB*. Morgan Kaufmann Publishers Inc., 1997, pp. 146–155.

[9] S. Chaudhuri and V. Narasayya, "Autoadmin ẅhat-ifïndex analysis utility," in *SIGMOD*. ACM, 1998, pp. 367–378.

[10] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for microsoft sql server 2005: demo," in *SIGMOD*, 2005, pp. 930–932.

[11] N. Bruno and S. Chaudhuri, "Automatic physical database tuning: a relaxation-based approach," in *SIGMOD*. ACM, 2005, pp. 227–238.

[12] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 24–43, 2005.

[13] A. Rasin and S. Zdonik, "An automatic physical design tool for clustered column-stores," in *EDBT*. ACM, 2013, pp. 203–214.

[14] S. Papadomanolakis and A. Ailamaki, "An integer linear programming approach to database design," in *IEEE ICDE*. IEEE, 2007, pp. 442–449.

[15] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik, "Coradd: correlation aware database designer for materialized views and indexes," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1103–1113, Sep. 2010.

[16] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*. ACM, 2006, pp. 671–682.

[17] C. Garcia-Alvarado, V. Raghavan, S. Narayanan, and F. M. Waas, "Automatic data placement in MPP databases," *ICDEW*, vol. 0, pp. 322–327, 2012.

[18] S. Rizzi and E. Saltarelli, "View materialization vs. indexing: Balancing space constraints in data warehouse design," in *Advanced Information Systems Engineering*. Springer, 2003, pp. 502–519.

[19] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007.

[20] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear, "Materialization strategies in the vertica analytic database: Lessons learned," in *ICDE*, 2013, pp. 1196–1207.

[21] "Intel Xeon X5670 Processor," http://ark.intel.com/products/47920/Intel-Xeon-Processor-X5670-(12M-Cache-2_93-GHz-6_40-GTs-Intel-QPI).